JOURNAL OF COMPUTER AND SYSTEM SCIENCES 10, 317–339 (1975)

Two-Way Nested Stack Automata Are Equivalent to Two-Way Stack Automata

C. BEERI

Institute of Mathematics, Hebrew University, Jerusalem, Israel

Received December 8, 1973

The computing power of 2-way Nested Stack Automata is investigated. It is shown that for any of the following types, a 2-way Nested Stack Automaton is equivalent to some 2-way Stack Automaton of the same type: Deterministic, Nondeterministic, k-head, $L(n)$-tape-bounded Auxiliary.

INTRODUCTION

An important part of Automata Theory has been concerned with the investigation of Automata which have a memory restricted in some way. The restriction may be global, like bounds on the time and tape, or it may be local, a restriction on the way the memory is manipulated. A well-known example of a machine of the second type is the Pushdown Automaton. A more powerful machine of this type is the Stack Automaton. The various 2-way Stack Automata have been extensively studied and complete characterizations have been given for them in terms of time and tape complexity classes [1, 6, 7].

In 1969, a still more powerful type of memory has been introduced—the Nested Stack [4]. It has been shown since that the 1-way Nested Stack Automata are strictly more powerful than the 1-way Stack Automata [8]. On the other hand there is practically no result on the computing power of the 2-way Nested Stack Automata. In this paper we show that nesting does not add to the computing power of the 2-way Stack Automata. That is, each of the various types of 2-way Nested Stack Automata is equivalent to the Stack Automata of the same type.

The key to the results is a technique for constructing, for a given 2-way Nested Stack Automaton, an equivalent machine, of the same type, which always halts. A similar construction for 2-way Stack Automata was given in [2]. We assume the reader is familiar with [2], so we concentrate on those details in which the construction for the Nested Stack differs from the one given in [2].

All the machines in this paper are 2-way so, from now on, we shall omit the "2-way".

1. Definitions and Preliminaries

In this section we give the basic definitions concerning the Nested Stack Automaton and related machines. For motivation and more details see [4]. Our definition differs from the one given by Aho in some minor details, but the machines are clearly equivalent.

Definition 1.1. A *Nondeterministic Nested Stack Automaton* (NSA) is a 7-tuple

$$S = \langle Q, \Sigma, \Gamma, q_0, \delta, \delta_\$, F\rangle,$$

where: Q, Σ, Γ are finite sets of internal states, input symbols, and stack symbols, respectively. The set Σ contains the distinguished symbols ¢ and \$, the left and right endmarkers, and each input is assumed to be a word in $¢(\Sigma - \{¢, \$\})^*\$$. The set Γ contains the distinguished symbols #, ¢, \$, where # serves as a bottom of stack marker, ¢ serves as a secondary bottom of stack marker, and \$ serves as a top of stack marker. $q_0 \in Q$ is the initial state and $F \subseteq Q$ is the set of final states.

The functions δ and $\delta_\$$ contain the table of moves of the machine; the first describes moves when the stack head is within the stack, the second describes moves when the stack head scans a \$ ("is at the top of a stack").

Thus, δ maps $Q \times \Sigma \times \Gamma$ to subsets of $(Q \times D \times D) \cup (Q \times D \times \{\$\})$ (where $D = \{-1, 0, 1\}$). $(q, d, x) \in \delta(p, a, z)$ means that when in state p, scanning a on the input and z ($z \neq \$$) on a stack, the machine may, in one move, perform the following operations.

(1) Change internal state to q.

(2) Move the input head d squares to the right (subject to the restriction that the input head cannot move off the input), where $d = -1$ means "move left".

(3) Change the stack and the position of the stack head according to the value of x:

(a) If $x \in D$ then the stack head moves x steps to the right, but not left of #. (The bottom of the stack is assumed to be at the left. We shall usually say "y is below z" instead of "y is to the left of z" and "y is above z" instead of "y is to the right of z"). In that case we say that the machine is in a stack reading mode (see [4]).

(b) If $x = \$$ then a new stack is created below z by inserting ¢\$ just below (to the left of) z. The stack head now scans the top of the new stack (the new \$). This is a stack creation mode. In this case z must be different from #, ¢; no new stack can be created below #, ¢.

If the stack head scans a \$ then the table of moves is given by $\delta_\$$. The move depends on the symbol $z \in \Gamma - \{\$\}$ immediately to the left of the scanned \$.

Formally, $\delta_\$$ is a function from $Q \times \Sigma \times \Gamma$ to subsets of $Q \times D \times (\{-1, 0, E\} \cup (\Gamma - \{\#, ¢, \$\}))$. $(q, d, x) \in \delta_\$(p, a, z)$ means that when in state p, scanning a on the input and $z\$$ on the stack, the machine may, in one move, perform the following operations.

(1) Change the internal state to q.

(2) Move the input head d steps to the right (but not off the input).

(3) Change the stack and the stack head position as follows.

(a) If $x = -1$ the stack head moves one step left (stack reading mode).

(b) If $x = 0$ the stack head remains stationary (pushdown mode).

(c) If $x = E$ we have two subcases: If $z \neq ¢, \#$, then $z\$$ is erased, $\$$ is printed, and the stack head scans this $\$$ (pushdown mode). If $z = ¢$ (or $z = \#$) then the pair $¢\$$ ($\#\$$) is erased and the stack head moves to the symbol above (to the right of) the erased $\$$. (If $z = \#$ there is no such symbol, the stack is empty—the machine halts); this is a stack destruction mode.

(d) If $x = z' \in \Gamma - \{\#, ¢, \$\}$ then $z'\$$ is printed instead of $\$$ and the stack head scans $\$$ (pushdown mode).

We emphasize that no stack is ever created below any $\$$.

DEFINITION 1.2. Let $w = a_1 \cdots a_n \in \Sigma^*$ be a fixed input of length n. (From now on, whenever no input is mentioned, we assume that the input is this w.) A *w-configuration* of S (or just a configuration if w is understood) is a 4-tuple $C=(q, i, y, j)$ where $q \in Q$, $y \in \Gamma^*$, i, and j are integers, $1 \leqslant i \leqslant n$, $0 \leqslant j \leqslant |y|$. (For a string y, $|y|$ is the number of symbols in y.) We say that S is in configuration C if S is in state q, the input head position (counted from the left) is i, the contents of the stack is y, and the position of the stack head (counted from the right) is j. In particular, $j = 0$ iff $y = \epsilon$, the empty stack. The configuration $(q_0, 1, \#\$, 1)$ is called the *initial configuration.*

DEFINITION 1.3. Let C_1, C_2 be w-configurations. We write $C_1 \vdash_w C_2$ ($C_1 \vdash C_2$ if w is understood) if, on input w, S may pass from C_1 to C_2 according to δ or $\delta_\$$. We denote by $\overset{*}{\vdash}_w$ ($\overset{*}{\vdash}$) the reflexive transitive closure of $\vdash_w$. If $C_1 \overset{*}{\vdash}_w C_2$ where C_1 is the initial configuration we say that C_2 is accessible (for input w) by S.

DEFINITION 1.4. A *w-computation* (or just a computation) of S is a sequence $V = C_1, C_2, \cdots$ such that: C_1 is the initial configuration, $C_i \vdash_w C_{i+1}$ for $i \geqslant 1$ and if C_m is the last configuration in V then S in configuration C_m has no next move. (V may be infinite, of course.) If the first and third conditions are dropped, V is called a *partial computation.* If $V = C_1, \ldots, C_{m+1}$ then m is the *length* of V.

DEFINITION 1.5. *S accepts w by final state* if there exists a w-computation of S, $C_1, C_2, \ldots, C_m$, such that $C_m = (q_m, i_m, y_m, j_m)$ and $q_m \in F$. *S accepts w by empty stack* if there is a w-computation as above and $C_m = (q_m, i_m, \epsilon, 0)$.

It is known [4] that for NSA the two notions of acceptance are equivalent. In this paper acceptance is by empty stack and final state simultaneously. The set accepted by S is denoted $N(S)$.

If, for some S, $|\delta(p, a, z)| \leqslant 1$ and $|\delta_S(p, a, z)| \leqslant 1$ for all p, a, z, then the machine is called a Deterministic Nested Stack Automaton (DNSA). (For a set A, $|A|$ is the number of elements in A.)

If the machine can operate only in the pushdown and stack reading modes we have the familar Stack Automaton (SA, DSA).

We shall also discuss the following extensions.

(1) A k-head machine is a machine with k heads on the input. The heads are assumed to be independent and can pass each other freely. For more detailed definitions see [5, 7]. Thus we have: k-NSA, k-DNSA, k-SA, k-DSA. Whenever k is omitted we assume $k = 1$.

(2) If we add to our machine a Turing Machine work tape we get the Auxiliary Nested Stack Automaton. Thus we have: Aux. NSA, Aux. DNSA, Aux. SA, Aux. DSA. Machines of this type are discussed in [6, 7].

Let $L(n) \geqslant \log n$ be a function on the nonnegative integers. An Aux. NSA S is said to accept a set A within tape bound $L(n)$ if: (a) $A = N(S)$. (b) For each $w \in A$ there exists an accepting w-computation for which the length of the work tape is bounded by $L(|w|)$. (The length of the stack tape is not counted). Thus we have: $L(n)$ tape-bounded Aux. NSA, Aux. DNSA, Aux. SA, Aux. DSA.

2. STACKS, COMPUTATIONS AND TRANSITION MATRICES

It is clear from the rules of the NSA described in the preceding section that if y is the stack in some configuration accessible by S then y must fulfill certain conditions. We now describe those conditions. For brevity's sake we talk about "stacks" meaning "nested stacks".

Let $y = z_m \cdots z_1 \in \Gamma^*$ be given, where $z_1 = \$$ and $z_m \in \{¢, \#\}$. We define two sets of integers as follows.

$$¢(y) = \{i \mid z_i = ¢\} \cup \{m\}; \qquad \$(y) = \{i \mid z_i = \$\}.$$

Let f_y: $¢(y) \to \$(y)$ be the partial function defined by the following procedure:

(a) $F = \varnothing$

(b) Let i be the minimal element in $¢(y)$ for which there exists a $j \in \$(y), j < i$, such that there is no element of $¢(y) \cup \$(y) - F$ between i and j. Set $f_y(i) = j$. If there is no such i then Stop.

(c) $F = F \cup \{i, j\}$; go to (b).

DEFINITION 2.1. The set of stacks on Γ, denoted by $\mathrm{St}(\Gamma)$, is the set:

$$\mathrm{St}(\Gamma) = \{y \in \{¢, \#\}\, \Gamma^*\$ \mid f_y \text{ is one-to-one from } ¢(y) \text{ onto } \$(y)\}.$$

Clearly, any y which is the stack in some accessible configuration belongs to $\mathrm{St}(\Gamma)$, but the opposite is not necessarily true. (Note: $\mathrm{St}(\Gamma)$ contains elements of $\#\Gamma^*$ and of $¢\Gamma^*$ as well since we want to talk about subwords of a stack which have themselves the structure of a stack.) From now on, when we say stack, we mean an element of $\mathrm{St}(\Gamma)$. When we talk about a configuration $C = (q, i, y, j)$ we shall assume that $y \in \mathrm{St}(\Gamma)$. Also, if $y = z_m \cdots z_j \cdots z_1$, then $z_k \neq \$$ for all $k > j$. (The stack head can pass a $¢$ to the left but it can never pass a $ to the right.)

Suppose $y = z_m \cdots z_1 \in \mathrm{St}(\Gamma)$. If $f_y(i) = j$ then the subword of y, $z_i \cdots z_j$ is also a member of $\mathrm{St}(\Gamma)$. If $i < m$ we call it a *secondary* stack (relative to y). We call y the *principal* stack (relative to itself). Whenever a stack u is secondary relative to a stack v we say that u is *embedded* in v.

DEFINITION 2.2. The *length* of $y \in \mathrm{St}(\Gamma)$, denoted by $l(y)$, is the number of symbols in y which do not belong to any stack embedded in y. (Thus, $l(y)$ may be different from $|y|$.)

We turn now to the important notion of degree of nesting of a stack.

DEFINITION 2.3. Let $u_1, u_2, \ldots, u_l$ be a series of stacks (in some stack y) such that for all $1 \leqslant i < l$:

(1) u_{i+1} is embedded in u_i.

(2) There exists no stack v such that v is embedded in u_i and u_{i+1} is embedded in v.

Then *the degree of nesting* of u_l in u_1 is l. We write $dn(u_l, u_1) = l$. (In particular, $dn(u_1, u_1) = 1$.) If z belongs to u_l but not to any stack embedded in u_l then $dn(z, u_1) = l$, For $y \in \mathrm{St}(\Gamma)$ we define:

$$dn(y) = \max_{\substack{u \text{ embedded in } y \\ \cup\{y\}}} \{dn(u, y)\}.$$

For a configuration $C = (q, i, y, j)$ we define: $dn(C) = dn(y)$.

DEFINITION 2.4.

$$\mathrm{Sub}(\Gamma) = \{y \in (\Gamma - \$)^* \mid \exists u, x \in \Gamma^* \text{ s. t. } uyx \in \mathrm{St}(\Gamma)\}.$$

The elements of $\mathrm{Sub}(\Gamma)$ will be called ***partial stacks***. For $u = z_m \cdots z_1 \in \mathrm{Sub}(\Gamma)$, $dn(u) = dn(z_1) =$ number of ¢'s (including #) in u, if $z_m = ¢, \#$. If $z_m \neq ¢, \$$ then $dn(u) = dn(¢u)$.

Note that if $u \in \mathrm{Sub}(\Gamma)$ then $u\$ \cdots \$ \in \mathrm{St}(\Gamma)$ if u begins with a ¢ or # and $\#u\$ \cdots \$ \in \mathrm{St}(\Gamma)$ otherwise. The number of \$'s is $dn(u)$.

We want to single out certain types of partial computations.

DEFINITION 2.5. Let $V = C_1, C_2, \ldots, C_l, \ldots$ be a partial computation where $C_l = (q_l, i_l, y_l, j_l)$. We call V a *subcomputation* if:

(1) $y_1 = z_m \cdots z_{j_1} \cdots z_1$ and $z_{j_1+1} = ¢$, $z_{j_1} = \$$.

(2) If V is infinite then the stack head never moves right of z_{j_1}.

(3) If V is finite then in the next to last configuration the stack is again y_1 and on the passage to the last configuration the stack $z_{j_1+1}z_{j_1} = ¢\$$ is destroyed.

Intuitively, a subcomputation is a partial computation from the creation of a stack to its destruction (or to infinity if it is not destroyed). Of course, during a subcomputation, the stack head may go down below the new stack but never above it. We shall refer to the stack "created" in the subcomputation as the stack of the subcomputation.

A configuration $C = (q, i, y, j)$ where $y = z_m \cdots z_1$ is called a *saddle* if $z_j = \$$.

DEFINITION 2.6. A partial computation V is a *stack scan* if its first configuration is a saddle and either V is infinite and the stack head never scans the same \$ again or V is finite and V scans the same \$ again for the first time in the last configuration. V is assumed to contain some configuration which is not a saddle.

DEFINITION 2.7. Let $y \in \mathrm{Sub}(\Gamma)$. A partial computation is called a *computation on* y if:

(1) The stack of the first configuration is uyx for some $u, x \in \Gamma^*$.

(2) If V is infinite then the stack head never visits u or x.

(3) If V is finite then the stack head first enters u or x in the last configuration of V.

A computation on y is a computation such that the stack head does not leave the region first occupied by y, except in the last configuration if it exists. Since $y \in \mathrm{Sub}(\Gamma)$, y contains no occurrences of \$, so no symbol of y can be erased unless the stack head first leaves y to the right. Thus the region first occupied by y is well defined. Symbols may be added to it, of course, by creating new stacks (including the case where a new stack is created just below y, the symbols of which belong to the region) but none of the original symbols is deleted. We note that the computation on y does not depend on u and x. Thus, from now on, when talking about a computation on y, we put in the stack component of configurations only the region which the stack head may visit. The first configuration is written as (q, i, y, j) instead of $(q, i, uyx, j + |x|)$, and so on. By the depth of nesting of such a configuration we shall mean the depth of nesting of the region of the stack actually written. Of course, such a "stack" can be easily made a stack by adding a string of \$'s on the right and a # on the left, if necessary.

DEFINITION 2.8. A computation on $y \in \mathrm{Sub}(\Gamma)$ is *a scan on* y if the first configuration is $(q, i, y, 1)$ and if it is finite then the stack head leaves y on the right side.

We turn now to the description of elements of $\mathrm{Sub}(\Gamma)$ by transition matrices. A similar description was given in [2] for SA. For convenience we shall call a pair of the form (q, j), where $q \in Q$, $1 \leqslant j \leqslant n$ $(n = |w|)$, a *sip* (state-input pair). We start with the nondeterministic case.

DEFINITION 2.9. An *n-transition matrix* (n-t.m.) is an $n \times n$ matrix $M = (m_{ij})$ such that each m_{ij} is a subset of $Q \times Q$. (n will usually be omitted.)

Obviously, the number of n-transition matrices is $2^{|Q|^2 \cdot n^2}$.

DEFINITION 2.10. Let $y \in \mathrm{Sub}(\Gamma)$. M is the *transition matrix* of y if the following condition holds: For each i, j, $(p, q) \in m_{ij}$ iff there is a finite scan on y such that the first sip in the scan is (p, i) and the last sip (after the stack head moved off y to the right) is (q, j).

Intuitively, M is the t.m. of y if M contains the information about any possible passage from (p, i) to (q, j) (for any p, q, i, j) by a scan on y. There is one t.m. that is of importance to us, namely the t.m. of the empty stack, ϵ, defined by $M_\epsilon(p, i) = \varnothing$ for all (p, i).

DEFINITION 2.11. Let $y \in \mathrm{Sub}(\Gamma)$, M a t.m. An *M-computation* on y is a sequence of configurations $C_1, C_2, \ldots$ such that for each $i \geqslant 1$ one of the following two conditions holds.

(1) $C_i \vdash C_{i+1}$ (on y).

(2) In C_i the stack head scans the leftmost symbol of the y region and there is a possible move of the stack head to the left by which the machine enters sip(p, k). In C_{i+1} the stack head still scans the same symbol and the sip is (q, j). In M, $(p, q) \in m_{kj}$.

Intuitively, the definition means as follows: Suppsoe M is the t.m. of some y'. Then an M-computation on y is a simulation of a computation on $y'y$. In the y region the simulation is step by step. The simulation of a scan on y' is represented by one "move" (Case 2, above). In particular if a stack is created just below y then the subcomputation is reproduced step by step (except the scans on y', of course.)

In the following, if M is the t.m. of y' we shall not distinguish between the M-computation on y and the computation on $y'y$ that is being simulated. It should be noted, however, that the definition is valid even if M is not the t.m. of any y'.

There are two cases where an M-computation on y is just a computation on y. One is when $M = M_\varphi$. The second is when $y \in \#\Gamma^*$, since no move left of $\#$ is possible.

DEFINITION 2.12. An *M-scan on y* is an M-computation on y in which the stack head starts at the right side of y and, if the computation is finite, the stack head moves right off y in the last step.

DEFINITION 2.13. An M-subcomputation is a sequence of configurations C_1, C_2,... such that:

(1) $C_1 = (q, l, ¢\$, 1)$ (or $(q, l, \#\$, 1)$).

(2) If the sequence is finite then the stack in the last configuration is ϵ.

(3) For each i one of the following conditions holds:

(a) $C_i \vdash C_{i+1}$.

(b) In C_i the stack head scans the leftmost symbol of y (where y is the stack of the subcomputation) and there is a possible move of the stack head left by which the machine enters sip(p, k). In C_{i+1} the stack head scans the same symbol and the sip is (q, j). In M, $(p, q) \in m_{kj}$.

The remarks after Definition 2.11 apply here as well.

DEFINITION 2.14. Let $y \in \mathrm{Sub}(\Gamma)$, M a t.m. The t.m. of y *relative to M* (the M-t.m. of y) is the matrix $N = (n_{ij})$ where: $(p, q) \in n_{ij}$ iff there is an M-scan on y in which the first sip is (p, i) and the last sip is (q, j).

Clearly, if M is the t.m. of y' then N is the t.m. of $y'y$.

Suppose now that S is a DNSA. The only matrices of interest are those in which:

(1) For each i there is at most one j such that $m_{ij} \neq \varnothing$.

(2) For each i, j, $| m_{ij} | \leqslant 1$.

A matrix of this type is, as a matter of fact, a function:

$$M\colon\ Q \times \{1, 2,..., n\} \to Q \times \{1, 2,..., n\} \cup \{\varphi\},$$

where

$$M(p, i) = (q, j) \Leftrightarrow (p, q) \in m_{ij},$$

$$M(p, i) = \varphi \Leftrightarrow \quad \text{For each } j, \quad m_{ij} = \varnothing.$$

All the definitions given for transition matrices can now be rephrased for transition functions. We shall use the letters M, N,... for t.m.s and f, g, h,... for t.f.s.

The following fact is important. The number of t.f.s is $(|Q| \cdot n + 1)^{Q \cdot n}$, so there exists an integer b such that this number is less than $(bn)^{bn}$.

We turn now to the discussion of computations where the depth of nesting is bounded. Definitions will be given for NSA only, the definitions for DNSA being identical. In the following, wherever l is a bound on the depth of nesting, l is assumed to belong to $\{1, 2, ...\} \cup \{\infty\}$.

DEFINITION 2.15. Let $y \in \mathrm{Sub}(\Gamma)$. A *computation of depth l on y* is a computation on y, namely $C_1, C_2, ...$, such that, for all i, $dn(C_i) \leqslant l$. (By our conventions we write in the configurations only the y region and the depth of nesting of C_i is the depth of nesting of the stack actually written in C_i). From now on we shall write "an (l)-computation" meaning "a computation of depth l". (The parentheses distinguish this notation from the previously defined "f-computation" where f is a t.f.) *A scan of depth l on y* (an (l)-scan on y) is an (l)-computation on y which is a scan on y.

It should be noted in the definition above that if $dn(y) = j$ then $l \geqslant j$.

DEFINITION 2.16. A *subcomputation of depth l* (an (l)-subcomputation) is a subcomputation such that the depth of nesting of the stack of the computation does not exceed l. (Note that we are not concerned about what happens below the stack of the subcomputation where the depth of nesting may exceed l.)

DEFINITION 2.17. Let $y \in \mathrm{Sub}(\Gamma)$. The (l)-t.m. *of* y is the matrix $N = (n_{ij})$ where: $(p, q) \in n_{ij}$ iff there is an (l)-scan on y starting in $\mathrm{sip}(p, i)$ and ending in $\mathrm{sip}(q, j)$.

Let M be a t.m. By changing in Definition 2.15 all occurrences of "computation" to "M-computation" we get a definition of an *M-computation of depth l on y* (an M–(l)-computation on y). Similarly we can define an M–(l)-scan *on* y, an *M–(l)-subcomputation* and M–(l)-t.m. *of* y. Note that the bound on the depth of nesting applies only to the stack that is actually given and we are not interested in what happens in the "imaginery" stack represented by M.

If N is the M–(l)-t.m. of y and M is the t.m. of some y' then N represents those scans on $y'y$ which are l-bounded on y. Similiarly, if M is the L–(j)-t.m. of y' then N represents those L-scans on $y'y$ which are j-bounded on y' and l-bounded on y. This line of reasoning can, of couse, be continued.

An important case which will arise later is the following: Suppose $y \in \mathrm{Sub}(\Gamma)$, $dn(y) = j$ and during an (l)-scan on y $(l \geqslant j)$ a new stack is created just below the rightmost symbol of y. The subcomputation thus initiated must be $(l - j)$-bounded. Also, any part of it in which the stack head is below the new stack must be (l)-bounded.

In particular, if $y = y_1 z$, where z is a symbol, the (l)-t.m. of y_1 being N, then the subcomputation must be an N–$(l - j)$-subcomputation.

We close this section with the observation that for SA most of the definitions above have no importance. The depth of nesting is always 1, there are no embedded stacks, each subcomputation is a complete computation, etc.

3. The Halting Problem

Definition 3.1. An NSA is *halting iff* for each input w, each w-computation is finite.

The following lemma is taken from [2] where it is proved for SA.

Lemma 3.1. *An* NSA *is halting* iff *for each input w there exist k_1, k_2, k_3 such that if $V = C_1, C_2, \ldots, C_m, \ldots$ is a w-computation, where $C_m = (q_m, i_m, y_m, j_m)$, then:*

(1) *For all m, $l(y_m) \leqslant k_1$.*

(2) *If $C_{m_1}, C_{m_2}, \ldots, C_{m_{k_2}}$, $m_1 < m_2 < \cdots < m_{k_2}$, are saddles with the stack head at the top of the principal stack and all have the same stack component y, then there exists an integer m, $m_1 < m < m_{k_2}$, such that $l(y_m) < l(y)$.*

(3) *There is no stack scan (on the principal stack) the length of which exceeds k_3.*

(Condition 3 can be replaced by the equivalent:

(3′) *There is no infinite stack scan.*)

Proof. The proof given in [2] applies to NSA as well so we omit the details.

Note. The lemma remains valid if we rephrase it to apply to subcomputations. That is, a subcomputation is finite iff the three conditions are fulfilled for the stack of the subcomputation. As a result, the Constructions A, B that we shall describe later apply to secondary stacks and subcomputations on them as well. The same holds for the modifications of Construction C described in later sections.

Ullman [2] has shown that for each SA (DSA) there exists an equivalent halting SA (DSA). The equivalent machine can be built from the original one by using three constructions which guarantee the fulfillment of conditions 1–3. Since we want to use similar constructions for the NSA, we shall describe them here briefly. For details see [1, 2].

The constructions are based on the following facts.

(1) Given a word of length n^2 on the stack (in a suitable format), an SA can copy the word onto the stack, with local changes if necessary. In particular, if the word is a configuration of an n^2 tape-bounded Turing Machine, the SA can print on the stack a next configuration of the machine. If the word is a number between 0 and

$2^{bn^2} - 1$ (for some fixed b) the number can be copied, decremented or incremented by 1 and tested for 0 and overflow.

(2) Given a block of (bn) unary blocks separated by * symbols, each of length not exceeding (bn), a DSA can copy the blocks onto the stack, with local changes if necessary. Such a block can represent a configuration of an n-$\log n$ tape-bounded Deterministic Turing Machine and the DSA can compute the next configuration of the machine. The block can also represent a number between 0 and $(bn)^{bn} - 1$ and the DSA can perform on it the same operations as above.

Let S be an SA (DSA). The proofs of the following claims can be found in [2].

CLAIM 1. *There exists an integer b such that if $w \in N(S)$, $|w| = n$ then there exists an accepting w-computation in which the length of the stack in any configuration is at most 2^{bn^2} $((bn)^{bn})$.*

Construction A. Add a track to the stack. On this track the new machine will count the number of stack symbols up to 2^{bn^2} $((bn)^{bn})$. The count for each new symbol appears beside the symbol. The machine halts if the count overflows. (The counting is in binary for the SA and base bn with digits in unary for the DSA.)

CLAIM 2. *If $w \in N(S)$ then there exists an accepting w-computation such that the number of saddles with the same stack, between which the top symbol is not erased, is at most $|Q| \cdot n$.*

Construction B. Add another track to the stack. On this track the new machine will count the number of visits to a stack symbol in a saddle configuration. The count for a symbol is printed beside the symbol and, if the symbol is erased, the count is also erased. If the count exceeds $|Q| \cdot n$ the machine halts.

Construction C. Add still another track to the stack. Whenever a new symbol is printed on the stack, the new machine will compute and print beside it the transition matrix (function) of the part of the stack up to and including the new symbol. Each new matrix can be computed from the preceeding one and the new symbol. (The matrix of the first symbol is computed using the matrix of the empty stack as "base" matrix.)

Using those matrices (functions) the new machine simulates stack scans of the old machine by just consulting the matrix and going back to the top. Thus the new machine has no infinite scans.

In the three constructions, whenever a symbol is erased, the information on the tracks beside it is also erased. This information does not influence the operation of the machine in any other way except those specified by the construction. Clearly A takes care of condition 1, B takes care of condition 2, and C takes case of condition 3. A simulating machine using A, B, C simultaneously is, therefore, halting.

It can be verified that claims 1, 2 are also valid for NSA (DNSA)*. Therefore, A and B can be applied to NSA (DNSA) (to the principal and all secondary stacks). With C, the situation is different. When we try to build a t.m. M from a given t.m. N and a symbol z we may, at some point, have to create a new stack below z. The subcomputation thus initiated may be infinite and the construction of M will not terminate.

We treat this difficulty in the next two sections, showing that a modified form of C can be performed for NSA (DNSA). We get an equivalent halting machine which happens to be an SA (DSA). The DNSA is treated first and in the following section we treat the nondeterministic case.

4. The Deterministic Case

Lemma 4.1. *Let S be a* DNSA. *There exists an integer s such that if $V = C_1, C_2, \ldots$ is a w-computation and, for some i, $dn(C_i) > (sn)^{sn}$ then V is infinite.*

Proof. Let y be the stack in a configuration of V. Each stack u in y can be characterized by a triple (f, p, j) where: f is the t.f. of the part of y below u and (p, j) is the sip at the time of creation of u. This triple determines the behavior of S on u. That is, if another stack v with the same triple is created during the computation then the subcomputation on v will be a simulation of the subcomputation on u. (The subcomputations may differ in stack scans which go below u and v. However, f guarantees that the entry and exist sips of such scans will be identical.)

The number of transition functions is less than $(bn)^{bn}$, for some b. Therefore the number of triples is no more than $(sn)^{sn}$ for a suitable s.

Now, if $dn(C_i) > (sn)^{sn}$ then there are in the stack of C_i stacks u and v, with u embedded in v, characterized by the same triple. This implies that during the computation on u another stack with the same triple will be created, and so on. The stacks $v, u, \ldots$, will never be destroyed and V is infinite. Q.E.D.

As a consequence of Lemma 4.1 we need only consider computations in which the depth of nesting is bounded by $(sn)^{sn}$.

In the following lemmas and in the next section we shall talk about SA which start with some initial data written on the stack. In such a case the SA accepts w if, starting with the initial data on the stack, it can empty the stack and enter a final state simultaneously.

For convenience in the presentation of the lemmas we introduce the following notation. *Let S be an* NSA (DNSA), *p, q be states of S, $1 \leqslant i, j \leqslant n$, $1 \leqslant l \leqslant 2^{dn^2}((sn)^{sn})$, and M be a* t.m. *Suppose that S has an M-(l)-subcomputation in which the first* sip *is (p, i) and the last* sip *is (q, j). We write this information as $S(l, q, j, M, p, i)$.*

* Cf. *note added in proof* at end of article.

LEMMA 4.2. *Let S be a* DNSA. *There exists a* DSA S_1 *such that*:

(1) S_1 *is halting.*

(2) *For any n-t.f., f, integers* $1 \leqslant i, j \leqslant n$, *and states* p, q, *the following is true*: *If* S_1 *starts a computation on* w *with* $\#j * q * f * p * i\$$ ($*$ is a separator and $\#$ is the bottom) *written on the stack then* S_1 *accepts* w *iff* $S(1, q, j, f, p, i)$.) (Let $Q = \{q_1 \cdots q_{|Q|}\}$. f consists of $|Q| \cdot n$ blocks separated by $*s$. If $f(q_j, i) = (q_m, l)$ then q_m followed by l in unary is written in the $(|Q| \cdot (i-1) + j)$th block.)

Proof. The idea of the proof is that if S is not allowed to create new stacks within the stack then it behaves almost like a DSA and can be simulated by one. We first define an intermediate DSA, S', operating as follows.

Stage 1. S' erases $p * i\$$ on the stack, transferring the input head to position i and recording p in memory. Then S' prints ¢\$ on top of the stack and passes to Stage 2.

Stage 2. S' simulates the moves of S on the new stack (with bottom ¢). S is supposed to start in sip(p, i). If S tries to create a new stack within the stack S' halts without accepting. If S', during a stack scan, moves the stack head below ¢ then S' consults f. If, according to f, S returns to ¢ in sip(t, k), S' does the same; otherwise, S' halts. If ever S erases ¢\$ then S' does the same and passes to Stage 3.

Stage 3. After the erasure of ¢\$ S is in sip(q', j'). S' erases f, compares (q, j) with (q', j'), and accepts on equality (accepting by empty stack and final state).

Suppose now that if S creates a stack entering sip(p, i) S has a f–(1)-subcomputation leading to the destruction of the stack, entering sip(q', j'). Clearly, S' can simulate this subcomputation and arrive at Stage 3. S' will accept only if $(q', j') = (q, j)$. Therefore S' fulfills condition 2 of the lemma. We now modify S' to get a halting machine.

The new machine, S_1, will use in Stage 2 constructions A, B, and a modified construction C. The modification of C is as follows: S_1 will print, beside each new stack symbol, the (1)-t.f. of the stack (including the new symbol) *relative to f* (cf. Definition 2.17). This can be easily done. S_1, at the end of Stage 1, copies f to the special track just below ¢ and then uses f and ¢ to compute the (1)-t.f. of ¢ relative to f. Each new matrix is then computed from the preceding one using the new symbol. We note that a (1)-t.f. is just a t.f. for a DSA and the construction of a new t.f. from an old one and a symbol is described in detail in [2].

Using Lemma 3.1 it can now be shown that S_1 is halting. Also, if S' has a finite computation, S_1 can simulate it. So, S_1 operates as required. Q.E.D.

LEMMA 4.3. *Let S be a* DNSA, *s the constant of Lemma* 4.1. *There exists a* DSA S_2 *such that*:

(1) *S_2 is halting.*

(2) *For any n-t.f., f, integers $1 \leqslant i, j \leqslant n$, $1 \leqslant l \leqslant (sn)^{sn}$, and states p, q, the following is true. If S_2 starts a computation on w with $\#l * j * q * f * p * i\$$ written on the stack then S_2 accepts iff $S(l, q, j, f, p, i)$* (l is written in base sn with digits in unary).

Proof. S_2 is similar to S_1 though more complicated. The difference between the two machines is in the construction of the transition functions. The proof consists of three parts. In the first part we describe the overall operation of S_2, in the second we describe how S_2 computes new transition functions and in the third we prove that S_2 operates as required.

Overall operation of S_2: S_2 has four additional tracks on the stack, three for A, B, C and a fourth for recording numbers which are going to be between 1 and l.

Stage 1. S_2 erases $p * i\$$, transferring the input head to position i and recording p in memory. Then S_2 checks l. If $l = 1$ then the t.f. f is copied, ¢\$ is printed, and a copy of S_1 is activated, starting from Stage 2 of the preceding lemma. If $l > 1$, S_2 copies f, l (on the appropriate tracks) prints ¢\$ and passes to Stage 2.

Stage 2. S_2 simulates S (from sip(p, i)), using constructions A, B, and a modification of C similar to the one described in the preceding lemma. That is, for each new stack symbol S_2 computes and prints on the third track the (l)-t.f. of the stack (including the new symbol) *relative to f*.

Suppose for the moment that S_2 can compute the required transition functions. Then S_2 simulates S in the following sense: S_2 simulates directly moves at the top of the stack. Whenever S starts a stack scan S_2 consults the most recent t.f. If, according to this t.f., S returns from the scan in sip(t, k) then S_2 continues the simulation in that sip. If S_2 does not return S_2 halts. If S ever erases ¢\$, S_2 does the same and passes to Stage 3.

Stage 3. S has erased the stack and is in sip(q', j'). S_2 accepts iff $(q, j) = (q', j')$.

We turn now to the problem of constructing transition functions. Suppose that h is to be constructed using the given t.f. g and the symbol z. The t.f. g is an (l)-t.f. relative to f of the stack and h is the (l)-t.f. of z relative to g.

The algorithm is, in principle, the same as in [2]. S_2 computes $h(p, i)$, for each (p, i), in some fixed order. For a given (p, i), S_2 constructs a sequence of sips.

$$(*) \qquad (p, i) = (p_1, i_1), (p_2, i_2), \ldots, (p_m, i_m), \ldots,$$

where (p_{m+1}, i_{m+1}) is related to (p_m, i_m) as follows. If S scans z in sip(p_m, i_m) then S has a g–(l)-computation on z such that the next time the stack head scans z, S is in sip(p_{m+1}, i_{m+1}). (Note that in a computation on z, if the stack head moves right of z, it will not scan z a next time; a next time exists if either a stationary move is made or if a stack is created below z and later destroyed.)

If for some m, S, when scanning z in (p_m, i_m), moves the stack head right entering (q, j) then (p_m, i_m) is the last member of $(*)$ and $h(p, i) = (q, j)$. If the length of the sequence exceeds $|Q| \cdot n$ then S is in a loop, the stack head will never move right $-h(p, i) = \varphi$. Also, if, for some m, (p_{m+1}, i_{m+1}) does not exist and the stack head has not moved right from z on (p_m, i_m) then $h(p, i) = \varphi$ (since this means that a stack has been created below z which is not going to be destroyed). Clearly, those are the only alternatives.

The passage from (p_m, i_m) to (p_{m+1}, i_{m+1}) can be in one of the following forms.

(1) S passes on z from (p_m, i_m) to (p_{m+1}, i_{m+1}) in one step. This possibility can be checked using δ.

(2) S in (p_m, i_m), scanning z, moves the stack head left, performs an f-(l)-scan and returns to z in (p_{m+1}, i_{m+1}). In that case (p_{m+1}, i_{m+1}) is computed using g. Similarly, if S moves left and does not return to z then $h(p, i) = \varphi$ and this can also be computed from g.

(3) S, scanning z, in (p_m, i_m), creates a new stack below z, entering sip(p_m', i_m'). If this stack is later destroyed then S returns to z in (p_{m+1}, i_{m+1}). If it is not destroyed then (p_{m+1}, i_{m+1}) does not exist and $h(p, i) = \varphi$. (This is where the construction of a t.f. for a Nested Stack differs from the construction for a Stack.) S_2 operates as follows.

(i) S_2 "guesses" (p_{m+1}, i_{m+1}). (S_2 will cycle through all possible "guesses" in a deterministic manner.) A special sign is put on the guess to indicate it is only a guess.

(ii) S_2 prints on top of the stack the string $¢(l - 1) * p_{m+1} * i_{m+1} * g * p_m' * i_m'\$$. ($l - 1$ is computed using l; see Section 3.)

(iii) S_2 starts on the new "stack" (the part above the new ¢) from the beginning. (The data on the stack are in the format required by the lemma.) When the computation on the new stack halts (we shall later prove that it always halts), then, if S is in accepting state, the guess is successful and (p_{m+1}, i_{m+1}) has been computed. Otherwise a new guess is made. If all guesses have been exhausted then (p_{m+1}, i_{m+1}) does not exist and $h(p, i) = \varphi$.

We now prove that S_2 fulfills the two conditions of the lemma. The proof is by induction on l. For $l = 1$, S_2 is actually S_1 and we have the proof of Lemma 4.2. Suppose now that S_2 fulfills conditions 1, 2, for any initial stack with first number $m < l$; we prove it for $m = l$.

Let us assume for a moment that, using the procedure given above, S_2 can compute for each stack its f-(l)-t.f. Then, since S_2 has A, B, and the modified C, S_2 always halts. (As we have remarked, Lemma 3.1 and Claims 1, 2 apply to subcomputations as well.) If S has a f-(l)-subcomputation leading to the destruction of the stack then S_2 can simulate it and arrive at Stage 3. Therefore S_2 accepts iff S (l, q, j, f, p, i), as required. It remains to check the above procedure.

The procedure for computing $h(p, i)$ consists of computing the sequence

$$(*) \qquad (p, i) = (p_1, i_1), (p_2, i_2), \ldots, (p_m, i_m), \ldots$$

and checking each (p_m, i_m) for a move right and, at the same time, checking for overflow. If, for each (p, i), S_2 can compute the sequence $(*)$ such that (p_{m+1}, i_{m+1}) stands in the appropriate relation (see above) to (p_m, i_m), then, clearly, $h(p, i)$ can be computed correctly. Thus we have only to show that (p_{m+1}, i_{m+1}) can always be found or shown not to exist.

The crucial point in the computation of (p_{m+1}, i_{m+1}) is when the passage from (p_m, i_m) to (p_{m+1}, i_{m+1}) is by a subcomputation where a new stack is created and later destroyed. Now, S_2 is siumulating an f-(l)-subcomputation and the f-(l)-t.f. at the point where the new stack was created is g so that subcomputation on the new stack must be a g-$(l-1)$-subcomputation. (We do not distinguish between the g-$(l-1)$-subcomputation and the actual subcomputation it simulates since they exist or do not exist simultaneously.) Therefore the "question" posed for S_2 (by writing $¢(l-1) * p_{m+1} * i_{m+1} * g * p_m' * i_m'\$$) is the correct one. By the induction hypothesis S_2 halts on the new "stack" and gives the correct answer. This means that (p_{m+1}, i_{m+1}) can be computed by S_2 (or shown not to exist). Thus the new f-(l)-t.f. can be computed, which proves the lemma for $m = l$. Q.E.D.

We can now prove the main result of this section.

THEOREM 4.1. *Let S be a* DNSA. *There exists a* DSA S_3 *such that*:

(1) *S_3 is halting.*
(2) $N(S_3) = N(S)$.

Proof. We describe briefly the operation of S_3. S is assumed to accept by empty stack and final state, the set of final states being $\{\bar{q}\}$. Let f_φ be the t.f. of the empty stack ϵ ($f_\varphi(p, i) = \varphi$ for all p, i.) S_3 prints on the stack the following string (where $n = |w|$ and at the beginning $i = 1$):

$$(sn)^{sn} * \bar{q} * i * f_\varphi * q_0 * 1\$.$$

A copy of S_2 is now activated. If it accepts so does S_3. If it does not accept (it always halts) S_3 tests to see if $i = n$, rejecting if the answer is positive. If $i < n$, i is incremented, the initial stack is changed accordingly and S_2 is started again.

Since S_2 always halts, S_3 will also halt on any input. Also, it is clear that $w \in N(S) \Leftrightarrow$ There is some i such that S has a computation on w starting in $(q_0, 1, \#\$, 1)$ and ending in $(q, i, \epsilon, 0)$, with depth of nesting bounded by $(sn)^{sn} \Leftrightarrow$ there is some $1 \leqslant i \leqslant n$ such that S_2, when started with $(sn)^{sn} * \bar{q} * i * f_\varphi * q_0 * 1\$$ on the stack will accept $w \Leftrightarrow w \in N(S_3)$.

(An f_φ-subcomputation is just a subcomputation since the stack head never moves below the stack.) Q.E.D.

5. The Nondeterministic Case

Lemma 5.1. *Let S be an* NSA. *There exists an integer d such that if $w \in N(S)$ then there exists an accepting w-computation $V = C_1, C_2, \ldots$ with $dn(C_i) \leqslant 2^{dn^2}$ $(n = |w|)$ for all i.*

*Proof**. Let y be the stack in some configuration occurring in an accepting computation. Each stack u in y can be characterized by a quintuple (M, p, i, q, j) where: M is the t.m. of the part of y below u, (p, i) is the sip at the time u was created and (q, j) is the sip at the time u is destroyed (right after the destruction).

Since the number of transition matrices is $2^{|Q|^2 \cdot n^2}$, there exists an integer d such that the number of such quintuples is no more than 2^{dn^2}.

Suppose now that in some accepting computation there exists a configuration C with $dn(C) > 2^{dn^2}$. Then, in the stack of C, there exist embedded stacks u and v with u embedded in v and u and v are characterized by the same quintuple. We can get a shorter accepting computation by dropping the part of the computation between the creation and destruction of v and putting in the part between the creation and destruction of u.

Let V be a shortest accepting computation, then in V the depth of nesting is, clearly, no more than 2^{dn^2}. Q.E.D.

As a consequence of the lemma we need consider only computations where the depth of nesting is bounded by 2^{dn^2}.

Lemma 5.2. *Let S be an* NSA. *There exists an* SA S_1 *such that*:

(1) *S_1 is halting.*

(2) *For any n-*t.m. *M, integers $1 \leqslant i, j \leqslant n$ and states p, q, the following is true. If S_1 starts a computation on w with $\#j * q * M * p * i\$$ written on the stack then S_1 accepts w* iff $S(1, q, j, M, p, i)$.

Proof. The proof proceeds along the same lines as the proof of Lemma 4.2. The only difference is that here we are simulating a nondeterministic machine. As a result the actions of S' and S_1 in Stage 2 are nondeterministic. Also the construction of a new t.m. from an old one and a symbol is performed as described in [2] for SA. Q.E.D.

Before proceeding with $l > 1$ we want to introduce a special type of Turing Machine which we call an Oracle Machine. (To be exact it is not the machine that is special but the use we make of it.)

Definition 5.1. A *T.M. with an Oracle* (an Oracle Machine) is a Nondeterministic T.M., T, such that:

* Cf. *note added in proof* at end of article.

(1) T has a single work tape with two tracks.

(2) T has a distinguished state called the query state. When T is in the query state T has two choices for the next move (which may be identical sometimes). We call them choice 1 and choice 2. When T is in any other state T has at most one choice for the next move. Thus T is nondeterministic only in the query state.

The interpretation that we want to give to the Oracle Machine is as follows. T is a D.T.M. which may ask questions from an Oracle. T asks a question by entering the query state. The question is: Does the string on the second track satisfy a certain condition? If the answer is yes, then choice 1 is taken and if the answer is no, choice 2 is taken.

Anyway, T, by definition is just a T.M. Therefore, the usual notions of a computation, length of a computation, $L(n)$-tape bounded computation, etc., all have the usual meaning. When talking about computations, we shall not assume the first configuration to be the initial configuration.

Definition 5.2. Let S be a NSA, T an Oracle Machine. A computation of T is called *an S-computation* if:

(1) Each time the query state is entered the contents of track 2 is $l * q * j * M * p * i$ where l, i, j are integers, $1 \leqslant i, j \leqslant n$, $1 \leqslant l \leqslant 2^{dn^2}$, M is an n-t.m. and p, q are states of S ($n = | w |$).

(2) The next move is choice 1 iff

$$S(l, q, j, M, p, i).$$

We note that for any configuration of T there is a unique S-computation starting in that configuration. This is so because T is nondeterministic only when it "asks questions" and the conditions in the definition imply that each question has a unique answer.

Lemma 5.3. *Let S be a given* NSA. *There exists an Oracle Machine T such that: If T is started with M, z, l on the first track (M is an n-*t.m., *$z \in \Gamma - \{¢, \#, \$\}$, $1 \leqslant l \leqslant 2^{dn^2}$) then T has a finite S-computation at the end of which the contents of the first track is the matrix N, where N is the M–(l)-*t.m. *of z. The amount of tape required for the computation is proportional to n^2.*

Proof. Ullman [2] describes a D.T.M. which, given M, Z, computes the M–(1)-t.m. of z for a given SA. Our machine works, in principle, in the same way. The difference stems from the fact that here we deal with NSA.

Let (q, j) be given. We define a sequence of sets (depending on (q, j)) as follows.

$$R_1{}^l = \{(q, j)\},$$
$$R_m{}^l = \{(p, k)/ \text{ there is a } M\text{–}(l)\text{-computation on } z \text{ starting in } \mathrm{sip}(q, j) \text{ such that } S \text{ scans } z \text{ in } \mathrm{sip}(p, k) \text{ in one of the first } m \text{ times } z \text{ is scanned}\}.$$

Clearly $R_m{}^l \subseteq R^l_{m+1}$ and $R^l_{|Q|\cdot n}$ is a fixed point. Also, $(p, i) \in N(q, j)$ iff there is a $\mathrm{sip}(p', i') \in R^l_{|Q|\cdot n}$ such that S in $\mathrm{sip}(p', i')$, scanning z, has a move right entering (p, i).

Now, in order to compute $N(q, j)$, our machine has to construct $R^l_{|Q|\cdot n}$ and check for the existence of (p', i') as above. We describe how $R^l_{|Q|\cdot n}$ is constructed. Suppose all elements of $R_m{}^l$ have been computed and are written in a sequence where the elements of $R_m{}^l - R^l_{m-1}$ appear last. Clearly, the elements of R^l_{m-1} cannot generate elements of R^l_{m+1} which do not already belong to $R_m{}^l$. Therefore we check only the elements of $R_m{}^l - R^l_{m-1}$. For each new sip generated we check to see if it is not a member of $R_m{}^l$ and if not we add it at the end of the list.

Let $(q', j') \in R_m{}^l - R^l_{m-1}$ now be given. There are three possibilities. (Of course, more than one can occur and T checks all.)

(1) The stack head does not move.
(2) The stack head moves left.

The construction of the new elements in R^l_{m+1} resulting from those two possibilities can be done using δ and M. The reader is referred to [2] for details.

(3) S in $\mathrm{sip}(q', j')$, scanning z, creates a new stack below z entering $\mathrm{sip}(s, k)$.

In that case T "guesses" a $\mathrm{sip}(p, i)$ (T will cover all possible guesses in a deterministic manner) and copies on the next track the string

$$(l - 1) * s * k * M * p * i.$$

Then T enters a query state. If the answer is yes, T adds (p, i) as a candidate for membership in R^l_{m+1}. In any case (yes or no) T erases the second track and proceeds to the next guess. When all guesses have been exhausted T knows that (s, k) has been checked and goes on to try another move from (q', j'). When all possible moves from (q', j') have been checked, T moves to the next element of $R_m{}^l$. T has, in addition, a counter, and when $R_m{}^l$ is exhausted the counter is incremented. Thus T can check if $R^l_{|Q|\cdot n}$ has been computed. After $R^l_{|Q|\cdot n}$ has been computed T checks each element in $R^l_{|Q|\cdot n}$ for a possible move to the right, thus computing $N(q, j)$. Then $R^l_{|Q|\cdot n}$ is erased and T starts computing another element of N.

Starting with M, z, l, T has a unique S-computation. This computation clearly does not require more than $c \cdot n^2$ tape, and is finite. The proof that T computes the required matrix N is the same as in [2]. (Note that $\log l \leqslant dn^2$, so T does need tape proportional to n^2 for l.) Q.E.D.

LEMMA 5.4. *Let S be an* NSA, *d the constant of Lemma* 5.1. *There exists an* SA, S_2, *such that*:

(1) *S_2 is halting.*

(2) *For any n*-t.m., *M, integers $1 \leqslant i, j \leqslant n$, $1 \leqslant l \leqslant 2^{dn^2}$, and states p, q the following is true: If S_2 starts a computation on w with $\#l * j * q * M * p * i\$$ written on the stack then S_2 has an accepting w-computation iff $S(l, q, j, M, p, i)$.*

Proof. S_2 is similar to the machine S_2 constructed in the proof of Lemma 4.3 except that S_2 is nondeterministic in Stage 2, since the simulated S is nondeterministic. The main difference between the two machines is in the way a new t.m. is constructed. We shall concentrate on that point.

S_2 has to build an (l)-t.m. relative to M of a stack yz, given M_1, the M–(l)-t.m. of y, z, and l. Let us denote the new matrix by N.

It is known [1, 2] that an SA can simulate a computation of an n^2-tape-bounded T.M. by constructing successive configurations of the T.M. Let T be the Oracle Machine of the preceding lemma. Given M_1, z, l, T has an n^2-tape-bounded S-computation which constructs N. S_2 will simulate this computation. S_2 starts by constructing the initial configuration of T with M, z, l on the first track. Then S_2 constructs successive configurations of T. It may happen that T enters the query state. By the construction of T given in the preceding lemma, when this happens, the contents of the second track are

$$(l-1) * q * j * M_1 * p * i.$$

If S_2 wants to construct the next configuration of T in the S-computation S_2 must find the answer to the question: (q) Is it true that $S(l-1, q, j, M_1, p, i)$? S_2 does that by printing $¢(l-1) * j * q * M_1 * p * i\$$ on top of the stack and starting afrech on the new data. When S_2 halts it has the answer to the above question and since the next move of T is determined by this answer, S_2 can construct the next configuration. (The new "stack" is erased in any case.) When T halts, the new matrix N has been computed.

We prove by induction on l that S_2 fulfills the conditions of the lemma. For $l = 1$, S_2 operates like S_1 and we have Lemma 5.2. Suppose the assertion is true for all $m < l$, we prove it for $m = l$.

By Lemma 5.3, if T starts with M, z, l the S-computation of T is finite and terminates with N on the first track. Each time T enters the query state the first number on the second track is $l - 1$. By induction hypothesis, S_2, starting with initial stack, the first number of which is $l-1$, always halts and gives the correct answer to the question (q). This means that for any configuration of T (query state or not) S_2 can construct the next configuration in the S-computation. Since the S-computation is finite S_2 will arrive at the last configuration and N will be computed. This proves that for each M_1, z, S_2 can compute the M_1–(l)-t.m. of z.

As a result, S_2, having constructions A, B, and the modified C, always halts (when $m = l$). Also, if $S(l, q', j', M, p, i)$ then S has a subcomputation in which the depth of nesting is bounded by l and for which Claims 1, 2 are true. (It can be shown that the constructions given in [2] as a proof for Claims 1, 2, apply to (l)-subcomputations, yielding (l)-subcomputations.) This subcomputation can be simulated by S_2 and S_2 will arrive at Stage 3, accepting only if $(q, j) = (q', j')$. Thus S_2 fulfills Condition 2. Q.E.D.

THEOREM 5.1. *Let S be an* NSA. *There exists an* SA S_3 *such that:*

(1) S_3 *is halting.*

(2) $N(S_3) = N(S)$.

Proof. The construction of S_3 is like that given in the proof of Theorem 4.1, except that the bound on the depth of nesting is 2^{dn^2} (instead of $(sn)^{sn}$). Q.E.D.

6. Other Types of Machines

In this section we extend the results of Sections 4, 5 to the other types of machines described in Section 1. Ibarra [7] has shown that, given an equivalence between a class of SA and a class of tape-bounded T.M. or Aux. PDM, a similar relation can be deduced for k-head SA and $L(n)$-tape-bounded Aux. SA. It can easily be seen that his techniques can be applied to NSA. Since the proofs are quite involved, we refer the reader to [7], giving here only the minor changes that are necessary to apply the proofs to NSA.

THEOREM 6.1. *Let S be a k-head* NSA (DNSA) *where* $k \geqslant 2$. *There exists a k-head* SA (DSA) S_1 *such that* $N(S_1) = N(S)$.

Proof. We note that Lemma 2.2 of [7] can be proved for k-head NSA (DSA). First we note that while the top of the stack in [7] is a blank, we use \$ as a top of a stack (thus making it possible to distinguish the top in a secondary stack). It can be assumed, however, that the stack head never stays on \$. Whenever \$ is scanned, the stack head moves left, keeping the fact that it is actually scanning a \$ in memory (in the form $(q, 0)$). Therefore, in the simulation described in the proof of [7, Lemma 2.2] a block $yzyz \cdots yz\$$ is printed for each z. When the block is erased, \$ is printed.

We must also describe what the simulating machine does when the original is in the stack creation and stack destruction modes, which do not exist in SA. We assume the simulating machine has its own bottom of stack, $\bar{¢}$.

Stack creation mode. In Step 4, Case 2 of the simulation add the possibility of creating a stack of the form $\bar{¢}y¢ \cdots y¢\$$. This new stack is printed using the procedure of Step 4, Case 1(c).

Stack destruction mode. In Step 4, Case 1(b), add the following: If $z = ¢$ (or $z = \#$) then the block $*^r \Delta *^s y¢ \cdots y¢\$$ (or $*^r \Delta *^s y\# \cdots y\#\$$) is erased and the stack head scans the block immediately to the right of the erased block (or halts if $z = \#$).

Now that the lemma has been proven for k-head NSA (DNSA) we have a NSA (DNSA) accepting a language. By our results this language is accepted by some SA (DSA). Using [7, Theorem 2.3] we conclude that the original language, $N(S)$, is accepted by a n^{2k} ($n^k \cdot \log n$) tape-bounded Aux. PDM. By [7] this means that it is accepted by a k-head SA (DSA). Q.E.D.

THEOREM 6.2. *Let S be an $L(n)$-tape-bounded* Aux. NSA (*for some* $L(n) \geqslant \log n$). *There exists an $L(n)$-tape-bounded* Aux. DSA S_1 *such that* $N(S_1) = N(S)$.

Proof. First note that Lemma 3.2 of [7] holds if we replace every occurrence of SA by an occurrence of NSA. We leave it to the reader to supply the necessary changes in the proof. Since an NSA can be simulated by an SA we can use [7, Theorem 3.2] to obtain a $2^{CL(n)}$-tape-bounded Aux. PDM accepting the same language. Using [7, Theorem 3.1] we get the required $L(n)$-tape-bounded Aux. DSA accepting the language $N(S)$. Q.E.D.

COROLLARY 6.1. *The following classes are equivalent.*

(1) *$L(n)$-tape-bounded* Aux. DSA.

(2) *$L(n)$-tape-bounded* Aux. SA.

(3) *$L(n)$-tape-bounded* Aux. DNSA.

(4) *$L(n)$-tape-bounded* Aux. NSA.

(5) *$2^{CL(n)}$-tape-bounded* Aux. PDM.

(6) *$2^{2^{CL(n)}}$-time-bounded* D.T.M.

Proof. Clearly the classes 1, 2, 4 form an ascending chain and so do the classes 1, 3, 4. Since we have proved that 4 is included in 1 we get the equivalence of 1–4. Equivalence with 5, 6 was proved in [7].

Note added in proof. The proof of claim 1 in [2] can be applied to NSA yielding the fact that the claim is true for the principal stack. However, in the proof a subcomputation is replaced by a shorter one. The scans in the new subcomputation do not fit in their new place and they must be replaced by other scans. To prove that the claim is true for all stacks in a computation one has to define a sequence of replacements each one dealing with deeper subcomputations and prove that it is finite. The same problem arises in the proof of Lemma 5.1 which, therefore, is not complete in the form given here. Complete proofs are given in [9].

References

1. J. E. Hopcroft and J. D. Ullman, Nonerasing stack languages, *J. Comp. Sys. Sci.* **1** (1967), 166–186.
2. J. D. Ullman, Halting stack automata, *J. Assoc. Comput. Mach.* **16** (1969), 550–563.
3. A. V. Aho, Indexed grammars: An extension of context free grammars, *J. Assoc. Comput. Mach.* **15** (1968), 647–671.
4. A. V. Aho, Nested stack automata, *J. Assoc. Comput. Mach.* **16** (1969), 383–406.
5. M. A. Harrison and O. H. Ibarra, Multi-tape and multi-head pushdown automata, *Inform. and Contr.* 13 (1968), 433–470.
6. S. A. Cook, Variations on pushdown machines, *J. Assoc. Comput. Mach.* **18** (1971), 4–18.
7. O. H. Ibarra, Characterizations of some tape and time complexity classes of turing machines in terms of multihead and auxiliary stack automata, *J. Comput. System Sci.* **5** (1971), 88–117.
8. W. F. Ogden, Intercalation theorems for pushdown store and stack languages, Ph.D. Dissertation, Stanford University, 1968.
9. C. Beeri, On the complexity of two problems in automata theory, Ph.D. thesis (in Hebrew), The Hebrew University of Jerusalem, 1975.